Rocky Enterprise Linux 9.2 Manual Pages on command 'keytool.1'

C:\>man keytool.1

keytool(1) Security Tools keytool(1)

NAME

keytool - Manages a keystore (database) of cryptographic keys, X.509 certificate chains, and trusted certificates.

SYNOPSIS

keytool [commands]

commands

See Commands. These commands are categorized by task as follows:

? Create or Add Data to the Keystore

? -gencert

? -genkeypair

? -genseckey

? -importcert

? -importpassword

? Import Contents From Another Keystore

? -importkeystore

? Generate Certificate Request

? -certreq

? Export Data

? -exportcert

? Display Data

? -list

? -printcert

? -printcertreq

? -printcrl

? Manage the Keystore

? -storepasswd

? -keypasswd

? -delete

? -changealias

? Get Help

? -help

DESCRIPTION

The keytool command is a key and certificate management utility. It enables users to administer their own public/private key pairs and associated certificates for use in self-authentication (where the user authenticates himself or herself to other users and services) or data integrity and authentication services, using digital signatures. The keytool command also enables users to cache the public keys (in the form of certificates) of their communicating peers.

A certificate is a digitally signed statement from one entity (person, company, and so on.), that says that the public key (and some other information) of some other entity has a particular value. (See Certificate.) When data is digitally signed, the signature can be verified to check the data integrity and authenticity.

Integrity means that the data has not been modified or tampered with, and authenticity means the data comes from whoever claims to have created and signed it.

The keytool command also enables users to administer secret keys and passphrases used in symmetric encryption and decryption (DES).

The keytool command stores the keys and certificates in a keystore. See KeyStore aliases.

COMMAND AND OPTION NOTES

See Commands for a listing and description of the various commands.

? All command and option names are preceded by a minus sign (-).

? The options for each command can be provided in any order.

? All items not italicized or in braces or brackets are required to appear as is.

? Braces surrounding an option signify that a default value will be used when the option is not specified on the command line. See Option Defaults. Braces are also used around the -v, -rfc, and -J options, which only have meaning when they appear on the command line. They do not have any default values other than not existing.

? Brackets surrounding an option signify that the user is prompted for the values when the option is not specified on the command line. For the -keypass option, if you do not specify the option on the command line, then the keytool command first attempts to use the keystore password to recover the private/secret key. If this attempt fails, then the keytool command prompts you for the private/secret key password.

? Items in italics (option values) represent the actual values that must be supplied. For example, here is the format of the -printcert command:

keytool -printcert {-file cert_file} {-v}

When you specify a -printcert command, replace cert_file with the actual file name, as follows: keytool -printcert -file VScert.cer

? Option values must be put in quotation marks when they contain a blank (space).

? The -help option is the default. The keytool command is the same as keytool -help.

OPTION DEFAULTS

The following examples show the defaults for various option values.

-alias "mykey"

-keyalg

"DSA" (when using -genkeypair)

"DES" (when using -genseckey)

-keysize

2048 (when using -genkeypair and -keyalg is "RSA")

1024 (when using -genkeypair and -keyalg is "DSA")

256 (when using -genkeypair and -keyalg is "EC")

56 (when using -genseckey and -keyalg is "DES")

168 (when using -genseckey and -keyalg is "DESede")

-validity 90

-keystore <the file named .keystore in the user's home directory>

-storetype <the value of the "keystore.type" property in the
security properties file, which is returned by the static
getDefaultType method in java.security.KeyStore>

-file
stdin (if reading)
stdout (if writing)

-protected false

In generating a public/private key pair, the signature algorithm (-sigalg option)
is derived from the algorithm of the underlying private key:

? If the underlying private key is of type DSA, then the -sigalg option defaults to
SHA1withDSA.

? If the underlying private key is of type RSA, then the -sigalg option defaults to
SHA256withRSA.

? If the underlying private key is of type EC, then the -sigalg option defaults to
SHA256withECDSA.

For a full list of -keyalg and -sigalg arguments, see Java Cryptography
Architecture (JCA) Reference Guide at
http://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html#AppA

COMMON OPTIONS

The -v option can appear for all commands except -help. When the -v option appears,
it signifies verbose mode, which means that more information is provided in the
output.

There is also a -Jjavaoption argument that can appear for any command. When the
-Jjavaoption appears, the specified javaoption string is passed directly to the
Java interpreter. This option does not contain any spaces. It is useful for
adjusting the execution environment or memory usage. For a list of possible
interpreter options, type java -h or java -X at the command line.

These options can appear for all commands operating on a keystore:

-storetype storetype
This qualifier specifies the type of keystore to be instantiated.

-keystore keystore
The keystore location.
If the JKS storetype is used and a keystore file does not yet exist, then

certain keytool commands can result in a new keystore file being created. For example, if keytool -genkeypair is called and the -keystore option is not specified, the default keystore file named .keystore in the user's home directory is created when it does not already exist. Similarly, if the -keystore ks_file option is specified but ks_file does not exist, then it is created. For more information on the JKS storetype, see the KeyStore Implementation section in KeyStore aliases.

Note that the input stream from the -keystore option is passed to the KeyStore.load method. If NONE is specified as the URL, then a null stream is passed to the KeyStore.load method. NONE should be specified if the keystore is not file-based. For example, when it resides on a hardware token device.

-storepass[:env| :file] argument

The password that is used to protect the integrity of the keystore.

If the modifier env or file is not specified, then the password has the value argument, which must be at least 6 characters long. Otherwise, the password is retrieved as follows:

? env: Retrieve the password from the environment variable named argument.

? file: Retrieve the password from the file named argument.

Note: All other options that require passwords, such as -keypass, -srckeypass, -destkeypass, -srcstorepass, and -deststorepass, accept the env and file modifiers. Remember to separate the password option and the modifier with a colon (:).

The password must be provided to all commands that access the keystore contents. For such commands, when the -storepass option is not provided at the command line, the user is prompted for it.

When retrieving information from the keystore, the password is optional. If no password is specified, then the integrity of the retrieved information cannot be verified and a warning is displayed.

-providerName provider_name

Used to identify a cryptographic service provider's name when listed in the security properties file.

-providerClass provider_class_name

Used to specify the name of a cryptographic service provider's master class file when the service provider is not listed in the security properties

file.

-providerArg provider_arg

Used with the -providerClass option to represent an optional string input argument for the constructor of provider_class_name.

-protected

Either true or false. This value should be specified as true when a password must be specified by way of a protected authentication path such as a dedicated PIN reader.Because there are two keystores involved in the -importkeystore command, the following two options -srcprotected and -destprotected are provided for the source keystore and the destination keystore respectively.

-ext {name{:critical} {=value}}

Denotes an X.509 certificate extension. The option can be used in -genkeypair and -gencert to embed extensions into the certificate generated, or in -certreq to show what extensions are requested in the certificate request. The option can appear multiple times. The name argument can be a supported extension name (see Named Extensions) or an arbitrary OID number. The value argument, when provided, denotes the argument for the extension. When value is omitted, that means that the default value of the extension or the extension requires no argument. The :critical modifier, when provided, means the extension's isCritical attribute is true; otherwise, it is false. You can use :c in place of :critical.

NAMED EXTENSIONS

The keytool command supports these named extensions. The names are not case-sensitive).

BC or BasicContraints

Values: The full form is: ca:{true|false}[,pathlen:<len>] or <len>, which is short for ca:true,pathlen:<len>. When <len> is omitted, you have ca:true.

KU or KeyUsage

Values: usage(,usage)*, where usage can be one of digitalSignature, nonRepudiation (contentCommitment), keyEncipherment, dataEncipherment, keyAgreement, keyCertSign, cRLSign, encipherOnly, decipherOnly. The usage argument can be abbreviated with the first few letters (dig for

digitalSignature) or in camel-case style (dS for digitalSignature or cRLS for cRLSign), as long as no ambiguity is found. The usage values are case-sensitive.

EKU or ExtendedKeyUsage

Values: usage(,usage)*, where usage can be one of anyExtendedKeyUsage, serverAuth, clientAuth, codeSigning, emailProtection, timeStamping, OCSPSigning, or any OID string. The usage argument can be abbreviated with the first few letters or in camel-case style, as long as no ambiguity is found. The usage values are case-sensitive.

SAN or SubjectAlternativeName

Values: type:value(,type:value)*, where type can be EMAIL, URI, DNS, IP, or OID. The value argument is the string format value for the type.

IAN or IssuerAlternativeName

Values: Same as SubjectAlternativeName.

SIA or SubjectInfoAccess

Values: method:location-type:location-value (,method:location-type:location-value)*, where method can be timeStamping, caRepository or any OID. The location-type and location-value arguments can be any type:value supported by the SubjectAlternativeName extension.

AIA or AuthorityInfoAccess

Values: Same as SubjectInfoAccess. The method argument can be ocsp,caIssuers, or any OID.

When name is OID, the value is the hexadecimal dumped DER encoding of the extnValue for the extension excluding the OCTET STRING type and length bytes. Any extra character other than standard hexadecimal numbers (0-9, a-f, A-F) are ignored in the HEX string. Therefore, both 01:02:03:04 and 01020304 are accepted as identical values. When there is no value, the extension has an empty value field.

A special name honored, used in -gencert only, denotes how the extensions included in the certificate request should be honored. The value for this name is a comma separated list of all (all requested extensions are honored), name{:[critical|non-critical]} (the named extension is honored, but using a different isCritical attribute) and -name (used with all, denotes an exception). Requested extensions are not honored by default.

If, besides the-ext honored option, another named or OID -ext option is provided, this extension is added to those already honored. However, if this name (or OID) also appears in the honored value, then its value and criticality overrides the one in the request.

The subjectKeyIdentifier extension is always created. For non-self-signed certificates, the authorityKeyIdentifier is created.

Note: Users should be aware that some combinations of extensions (and other certificate fields) may not conform to the Internet standard. See Certificate Conformance Warning.

COMMANDS

-gencert

{-rfc} {-infile infile} {-outfile outfile} {-alias alias} {-sigalg sigalg} {-dname dname} {-startdate startdate {-ext ext}* {-validity valDays} [-keypass keypass] {-keystore keystore} [-storepass storepass] {-storetype storetype} {-providername provider_name} {-providerClass provider_class_name {-providerArg provider_arg}} {-v} {-protected} {-Jjavaoption}

Generates a certificate as a response to a certificate request file (which can be created by the keytool-certreq command). The command reads the request from infile (if omitted, from the standard input), signs it using alias's private key, and outputs the X.509 certificate into outfile (if omitted, to the standard output). When-rfc is specified, the output format is Base64-encoded PEM; otherwise, a binary DER is created.

The sigalg value specifies the algorithm that should be used to sign the certificate. The startdate argument is the start time and date that the certificate is valid. The valDays argument tells the number of days for which the certificate should be considered valid.

When dname is provided, it is used as the subject of the generated certificate. Otherwise, the one from the certificate request is used.

The ext value shows what X.509 extensions will be embedded in the certificate. Read Common Options for the grammar of -ext.

The -gencert option enables you to create certificate chains. The following example creates a certificate, e1, that contains three certificates in its

certificate chain.

The following commands creates four key pairs named ca, ca1, ca2, and e1:

```
keytool -alias ca -dname CN=CA -genkeypair
keytool -alias ca1 -dname CN=CA -genkeypair
keytool -alias ca2 -dname CN=CA -genkeypair
keytool -alias e1 -dname CN=E1 -genkeypair
```

The following two commands create a chain of signed certificates; ca signs ca1 and ca1 signs ca2, all of which are self-issued:

```
keytool -alias ca1 -certreq |
    keytool -alias ca -gencert -ext san=dns:ca1 |
    keytool -alias ca1 -importcert
keytool -alias ca2 -certreq |
    $KT -alias ca1 -gencert -ext san=dns:ca2 |
    $KT -alias ca2 -importcert
```

The following command creates the certificate e1 and stores it in the file e1.cert, which is signed by ca2. As a result, e1 should contain ca, ca1, and ca2 in its certificate chain:

```
keytool -alias e1 -certreq | keytool -alias ca2 -gencert > e1.cert
```

-genkeypair

{-alias alias} {-keyalg keyalg} {-keysize keysize} {-sigalg sigalg} [-dname dname] [-keypass keypass] {-startdate value} {-ext ext}* {-validity valDays} {-storetype storetype} {-keystore keystore} [-storepass storepass] {-providerClass provider_class_name {-providerArg provider_arg}} {-v} {-protected} {-Jjavaoption}

Generates a key pair (a public key and associated private key). Wraps the public key into an X.509 v3 self-signed certificate, which is stored as a single-element certificate chain. This certificate chain and the private key are stored in a new keystore entry identified by alias.

The keyalg value specifies the algorithm to be used to generate the key pair, and the keysize value specifies the size of each key to be generated. The sigalg value specifies the algorithm that should be used to sign the self-signed certificate. This algorithm must be compatible with the keyalg

value.

The dname value specifies the X.500 Distinguished Name to be associated with the value of alias, and is used as the issuer and subject fields in the self-signed certificate. If no distinguished name is provided at the command line, then the user is prompted for one.

The value of keypass is a password used to protect the private key of the generated key pair. If no password is provided, then the user is prompted for it. If you press the Return key at the prompt, then the key password is set to the same password as the keystore password. The keypass value must be at least 6 characters.

The value of startdate specifies the issue time of the certificate, also known as the "Not Before" value of the X.509 certificate's Validity field.

The option value can be set in one of these two forms:

([+-]nnn[ymdHMS])+

[yyyy/mm/dd] [HH:MM:SS]

With the first form, the issue time is shifted by the specified value from the current time. The value is a concatenation of a sequence of subvalues. Inside each subvalue, the plus sign (+) means shift forward, and the minus sign (-) means shift backward. The time to be shifted is nnn units of years, months, days, hours, minutes, or seconds (denoted by a single character of y, m, d, H, M, or S respectively). The exact value of the issue time is calculated using the java.util.GregorianCalendar.add(int field, int amount) method on each subvalue, from left to right. For example, by specifying, the issue time will be:

Calendar c = new GregorianCalendar();

c.add(Calendar.YEAR, -1);

c.add(Calendar.MONTH, 1);

c.add(Calendar.DATE, -1);

return c.getTime()

With the second form, the user sets the exact issue time in two parts, year/month/day and hour:minute:second (using the local time zone). The user can provide only one part, which means the other part is the same as the current date (or time). The user must provide the exact number of digits as

shown in the format definition (padding with 0 when shorter). When both the date and time are provided, there is one (and only one) space character between the two parts. The hour should always be provided in 24 hour format. When the option is not provided, the start date is the current time. The option can be provided at most once.

The value of valDays specifies the number of days (starting at the date specified by -startdate, or the current date when -startdate is not specified) for which the certificate should be considered valid.

This command was named -genkey in earlier releases. The old name is still supported in this release. The new name, -genkeypair, is preferred going forward.

-genseckey

{-alias alias} {-keyalg keyalg} {-keysize keysize} [-keypass keypass] {-storetype storetype} {-keystore keystore} [-storepass storepass] {-providerClass provider_class_name {-providerArg provider_arg}} {-v} {-protected} {-Jjavaoption}

Generates a secret key and stores it in a new KeyStore.SecretKeyEntry identified by alias.

The value of keyalg specifies the algorithm to be used to generate the secret key, and the value of keysize specifies the size of the key to be generated. The keypass value is a password that protects the secret key. If no password is provided, then the user is prompted for it. If you press the Return key at the prompt, then the key password is set to the same password that is used for the keystore. The keypass value must be at least 6 characters.

-importcert

{-alias alias} {-file cert_file} [-keypass keypass] {-noprompt} {-trustcacerts} {-storetype storetype} {-keystore keystore} [-storepass storepass] {-providerName provider_name} {-providerClass provider_class_name {-providerArg provider_arg}} {-v} {-protected} {-Jjavaoption}

Reads the certificate or certificate chain (where the latter is supplied in a PKCS#7 formatted reply or a sequence of X.509 certificates) from the file

cert_file, and stores it in the keystore entry identified by alias. If no file is specified, then the certificate or certificate chain is read from stdin.

The keytool command can import X.509 v1, v2, and v3 certificates, and PKCS#7 formatted certificate chains consisting of certificates of that type. The data to be imported must be provided either in binary encoding format or in printable encoding format (also known as Base64 encoding) as defined by the Internet RFC 1421 standard. In the latter case, the encoding must be bounded at the beginning by a string that starts with -----BEGIN, and bounded at the end by a string that starts with -----END.

You import a certificate for two reasons: To add it to the list of trusted certificates, and to import a certificate reply received from a certificate authority (CA) as the result of submitting a Certificate Signing Request to that CA (see the -certreq option in Commands).

Which type of import is intended is indicated by the value of the -alias option. If the alias does not point to a key entry, then the keytool command assumes you are adding a trusted certificate entry. In this case, the alias should not already exist in the keystore. If the alias does already exist, then the keytool command outputs an error because there is already a trusted certificate for that alias, and does not import the certificate. If the alias points to a key entry, then the keytool command assumes you are importing a certificate reply.

-importpassword

{-alias alias} [-keypass keypass] {-storetype storetype} {-keystore keystore} [-storepass storepass] {-providerClass provider_class_name {-providerArg provider_arg}} {-v} {-protected} {-Jjavaoption}

Imports a passphrase and stores it in a new KeyStore.SecretKeyEntry identified by alias. The passphrase may be supplied via the standard input stream; otherwise the user is prompted for it. keypass is a password used to protect the imported passphrase. If no password is provided, the user is prompted for it. If you press the Return key at the prompt, the key password is set to the same password as that used for the keystore. keypass must be

at least 6 characters long.

-importkeystore

{-srcstoretype srcstoretype} {-deststoretype deststoretype}

[-srcstorepass srcstorepass] [-deststorepass deststorepass] {-srcprotected}

{-destprotected}

{-srcalias srcalias {-destalias destalias} [-srckeypass srckeypass]}

[-destkeypass destkeypass] {-noprompt}

{-srcProviderName src_provider_name} {-destProviderName dest_provider_name}

{-providerClass provider_class_name {-providerArg provider_arg}} {-v}

{-protected} {-Jjavaoption}

Imports a single entry or all entries from a source keystore to a

destination keystore.

When the -srcalias option is provided, the command imports the single entry

identified by the alias to the destination keystore. If a destination alias

is not provided with destalias, then srcalias is used as the destination

alias. If the source entry is protected by a password, then srckeypass is

used to recover the entry. If srckeypass is not provided, then the keytool

command attempts to use srcstorepass to recover the entry. If srcstorepass

is either not provided or is incorrect, then the user is prompted for a

password. The destination entry is protected with destkeypass. If

destkeypass is not provided, then the destination entry is protected with

the source entry password. For example, most third-party tools require

storepass and keypass in a PKCS #12 keystore to be the same. In order to

create a PKCS #12 keystore for these tools, always specify a -destkeypass to

be the same as -deststorepass.

If the -srcalias option is not provided, then all entries in the source

keystore are imported into the destination keystore. Each destination entry

is stored under the alias from the source entry. If the source entry is

protected by a password, then srcstorepass is used to recover the entry. If

srcstorepass is either not provided or is incorrect, then the user is

prompted for a password. If a source keystore entry type is not supported in

the destination keystore, or if an error occurs while storing an entry into

the destination keystore, then the user is prompted whether to skip the

entry and continue or to quit. The destination entry is protected with the source entry password.

If the destination alias already exists in the destination keystore, then the user is prompted to either overwrite the entry or to create a new entry under a different alias name.

If the -noprompt option is provided, then the user is not prompted for a new destination alias. Existing entries are overwritten with the destination alias name. Entries that cannot be imported are skipped and a warning is displayed.

-printcertreq

{-file file}

Prints the content of a PKCS #10 format certificate request, which can be generated by the keytool-certreq command. The command reads the request from file. If there is no file, then the request is read from the standard input.

-certreq

{-alias alias} {-dname dname} {-sigalg sigalg} {-file certreq_file}

[-keypass keypass] {-storetype storetype} {-keystore keystore}

[-storepass storepass] {-providerName provider_name}

{-providerClass provider_class_name {-providerArg provider_arg}}

{-v} {-protected} {-Jjavaoption}

Generates a Certificate Signing Request (CSR) using the PKCS #10 format.

A CSR is intended to be sent to a certificate authority (CA). The CA authenticates the certificate requestor (usually off-line) and will return a certificate or certificate chain, used to replace the existing certificate chain (which initially consists of a self-signed certificate) in the keystore.

The private key associated with alias is used to create the PKCS #10 certificate request. To access the private key, the correct password must be provided. If keypass is not provided at the command line and is different from the password used to protect the integrity of the keystore, then the user is prompted for it. If dname is provided, then it is used as the subject in the CSR. Otherwise, the X.500 Distinguished Name associated with alias is used.

The sigalg value specifies the algorithm that should be used to sign the CSR.

The CSR is stored in the file certreq_file. If no file is specified, then the CSR is output to stdout.

Use the importcert command to import the response from the CA.

-exportcert

{-alias alias} {-file cert_file} {-storetype storetype} {-keystore keystore} [-storepass storepass] {-providerName provider_name} {-providerClass provider_class_name {-providerArg provider_arg}} {-rfc} {-v} {-protected} {-Jjavaoption}

Reads from the keystore the certificate associated with alias and stores it in the cert_file file. When no file is specified, the certificate is output to stdout.

The certificate is by default output in binary encoding. If the -rfc option is specified, then the output in the printable encoding format defined by the Internet RFC 1421 Certificate Encoding Standard.

If alias refers to a trusted certificate, then that certificate is output. Otherwise, alias refers to a key entry with an associated certificate chain. In that case, the first certificate in the chain is returned. This certificate authenticates the public key of the entity addressed by alias.

This command was named -export in earlier releases. The old name is still supported in this release. The new name, -exportcert, is preferred going forward.

-list

{-alias alias} {-storetype storetype} {-keystore keystore} [-storepass storepass] {-providerName provider_name} {-providerClass provider_class_name {-providerArg provider_arg}} {-v | -rfc} {-protected} {-Jjavaoption}

Prints to stdout the contents of the keystore entry identified by alias. If no alias is specified, then the contents of the entire keystore are printed.

This command by default prints the SHA1 fingerprint of a certificate. If the -v option is specified, then the certificate is printed in human-readable format, with additional information such as the owner, issuer, serial

number, and any extensions. If the -rfc option is specified, then the certificate contents are printed using the printable encoding format, as defined by the Internet RFC 1421 Certificate Encoding Standard.

You cannot specify both -v and -rfc.

-printcert

{-file cert_file | -sslserver host[:port]} {-jarfile JAR_file {-rfc} {-v} {-Jjavaoption}

Reads the certificate from the file cert_file, the SSL server located at host:port, or the signed JAR file JAR_file (with the -jarfile option and prints its contents in a human-readable format. When no port is specified, the standard HTTPS port 443 is assumed. Note that -sslserver and -file options cannot be provided at the same time. Otherwise, an error is reported. If neither option is specified, then the certificate is read from stdin.

When-rfc is specified, the keytool command prints the certificate in PEM mode as defined by the Internet RFC 1421 Certificate Encoding standard. See Internet RFC 1421 Certificate Encoding Standard.

If the certificate is read from a file or stdin, then it might be either binary encoded or in printable encoding format, as defined by the RFC 1421 Certificate Encoding standard.

If the SSL server is behind a firewall, then the -J-Dhttps.proxyHost=proxyhost and -J-Dhttps.proxyPort=proxyport options can be specified on the command line for proxy tunneling. See Java Secure Socket Extension (JSSE) Reference Guide at http://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/JSSERefGuide.html

Note: This option can be used independently of a keystore.

-printcrl

-file crl_ {-v}

Reads the Certificate Revocation List (CRL) from the file crl_. A CRL is a list of digital certificates that were revoked by the CA that issued them.

The CA generates the crl_ file.

Note: This option can be used independently of a keystore.

-storepasswd

[-new new_storepass] {-storetype storetype} {-keystore keystore}

[-storepass storepass] {-providerName provider_name}

{-providerClass provider_class_name {-providerArg provider_arg}}

{-v} {-Jjavaoption}

Changes the password used to protect the integrity of the keystore contents.

The new password is new_storepass, which must be at least 6 characters.

-keypasswd

{-alias alias} [-keypass old_keypass] [-new new_keypass] {-storetype storetype}

{-keystore keystore} [-storepass storepass] {-providerName provider_name}

{-providerClass provider_class_name {-providerArg provider_arg}} {-v}

{-Jjavaoption}

Changes the password under which the private/secret key identified by alias

is protected, from old_keypass to new_keypass, which must be at least 6

characters.

If the -keypass option is not provided at the command line, and the key

password is different from the keystore password, then the user is prompted

for it.

If the -new option is not provided at the command line, then the user is

prompted for it

-delete

[-alias alias] {-storetype storetype} {-keystore keystore} [-storepass storepass]

{-providerName provider_name}

{-providerClass provider_class_name {-providerArg provider_arg}}

{-v} {-protected} {-Jjavaoption}

Deletes from the keystore the entry identified by alias. The user is

prompted for the alias, when no alias is provided at the command line.

-changealias

{-alias alias} [-destalias destalias] [-keypass keypass] {-storetype storetype}

{-keystore keystore} [-storepass storepass] {-providerName provider_name}

{-providerClass provider_class_name {-providerArg provider_arg}} {-v}

{-protected} {-Jjavaoption}

Move an existing keystore entry from the specified alias to a new alias,

destalias. If no destination alias is provided, then the command prompts for

one. If the original entry is protected with an entry password, then the password can be supplied with the -keypass option. If no key password is provided, then the storepass (if provided) is attempted first. If the attempt fails, then the user is prompted for a password.

-help

Lists the basic commands and their options.

For more information about a specific command, enter the following, where command_name is the name of the command: keytool -command_name -help.

EXAMPLES

This example walks through the sequence of steps to create a keystore for managing public/private key pair and certificates from trusted entities.

GENERATE THE KEY PAIR

First, create a keystore and generate the key pair. You can use a command such as the following typed as a single line:

keytool -genkeypair -dname "cn=Mark Jones, ou=Java, o=Oracle, c=US"
-alias business -keypass <new password for private key>
-keystore /working/mykeystore
-storepass <new password for keystore> -validity 180

The command creates the keystore named mykeystore in the working directory (assuming it does not already exist), and assigns it the password specified by <new password for keystore>. It generates a public/private key pair for the entity whose distinguished name has a common name of Mark Jones, organizational unit of Java, organization of Oracle and two-letter country code of US. It uses the default DSA key generation algorithm to create the keys; both are 1024 bits.

The command uses the default SHA1withDSA signature algorithm to create a self-signed certificate that includes the public key and the distinguished name information. The certificate is valid for 180 days, and is associated with the private key in a keystore entry referred to by the alias business. The private key is assigned the password specified by <new password for private key>.

The command is significantly shorter when the option defaults are accepted. In this case, no options are required, and the defaults are used for unspecified options that have default values. You are prompted for any required values. You could have the following:

keytool -genkeypair

In this case, a keystore entry with the alias mykey is created, with a newly generated key pair and a certificate that is valid for 90 days. This entry is placed in the keystore named .keystore in your home directory. The keystore is created when it does not already exist. You are prompted for the distinguished name information, the keystore password, and the private key password.

The rest of the examples assume you executed the -genkeypair command without options specified, and that you responded to the prompts with values equal to those specified in the first -genkeypair command. For example, a distinguished name of cn=Mark Jones, ou=Java, o=Oracle, c=US).

REQUEST A SIGNED CERTIFICATE FROM A CA

Generating the key pair created a self-signed certificate. A certificate is more likely to be trusted by others when it is signed by a Certification Authority (CA).

To get a CA signature, first generate a Certificate Signing Request (CSR), as follows:

keytool -certreq -file MarkJ.csr

This creates a CSR for the entity identified by the default alias mykey and puts the request in the file named MarkJ.csr. Submit this file to a CA, such as VeriSign. The CA authenticates you, the requestor (usually off-line), and returns a certificate, signed by them, authenticating your public key. In some cases, the CA returns a chain of certificates, each one authenticating the public key of the signer of the previous certificate in the chain.

IMPORT A CERTIFICATE FOR THE CA

You now need to replace the self-signed certificate with a certificate chain, where each certificate in the chain authenticates the public key of the signer of the previous certificate in the chain, up to a root CA.

Before you import the certificate reply from a CA, you need one or more trusted certificates in your keystore or in the cacerts keystore file. See -importcert in Commands.

? If the certificate reply is a certificate chain, then you need the top certificate of the chain. The root CA certificate that authenticates the public key of the CA.

? If the certificate reply is a single certificate, then you need a certificate for

the issuing CA (the one that signed it). If that certificate is not self-signed, then you need a certificate for its signer, and so on, up to a self-signed root CA certificate.

The cacerts keystore file ships with several VeriSign root CA certificates, so you probably will not need to import a VeriSign certificate as a trusted certificate in your keystore. But if you request a signed certificate from a different CA, and a certificate authenticating that CA's public key was not added to cacerts, then you must import a certificate from the CA as a trusted certificate.

A certificate from a CA is usually either self-signed or signed by another CA, in which case you need a certificate that authenticates that CA's public key. Suppose company ABC, Inc., is a CA, and you obtain a file named ABCCA.cer that is supposed to be a self-signed certificate from ABC, that authenticates that CA's public key.

Be careful to ensure the certificate is valid before you import it as a trusted certificate. View it first with the keytool -printcert command or the keytool -importcert command without the -noprompt option, and make sure that the displayed certificate fingerprints match the expected ones. You can call the person who sent the certificate, and compare the fingerprints that you see with the ones that they show or that a secure public key repository shows. Only when the fingerprints are equal is it guaranteed that the certificate was not replaced in transit with somebody else's (for example, an attacker's) certificate. If such an attack takes place, and you did not check the certificate before you imported it, then you would be trusting anything the attacker has signed.

If you trust that the certificate is valid, then you can add it to your keystore with the following command:

keytool -importcert -alias abc -file ABCCA.cer

This command creates a trusted certificate entry in the keystore, with the data from the file ABCCA.cer, and assigns the alias abc to the entry.

IMPORT THE CERTIFICATE REPLY FROM THE CA

After you import a certificate that authenticates the public key of the CA you submitted your certificate signing request to (or there is already such a certificate in the cacerts file), you can import the certificate reply and replace your self-signed certificate with a certificate chain. This chain is the one returned by the CA in response to your request (when the CA reply is a chain), or

one constructed (when the CA reply is a single certificate) using the certificate reply and trusted certificates that are already available in the keystore where you import the reply or in the cacerts keystore file.

For example, if you sent your certificate signing request to VeriSign, then you can import the reply with the following, which assumes the returned certificate is named VSMarkJ.cer:

keytool -importcert -trustcacerts -file VSMarkJ.cer

EXPORT A CERTIFICATE THAT AUTHENTICATES THE PUBLIC KEY

If you used the jarsigner command to sign a Java Archive (JAR) file, then clients that want to use the file will want to authenticate your signature. One way the clients can authenticate you is by first importing your public key certificate into their keystore as a trusted entry.

You can export the certificate and supply it to your clients. As an example, you can copy your certificate to a file named MJ.cer with the following command that assumes the entry has an alias of mykey:

keytool -exportcert -alias mykey -file MJ.cer

With the certificate and the signed JAR file, a client can use the jarsigner command to authenticate your signature.

IMPORT KEYSTORE

The command importkeystore is used to import an entire keystore into another keystore, which means all entries from the source keystore, including keys and certificates, are all imported to the destination keystore within a single command. You can use this command to import entries from a different type of keystore. During the import, all new entries in the destination keystore will have the same alias names and protection passwords (for secret keys and private keys). If the keytool command cannot recover the private keys or secret keys from the source keystore, then it prompts you for a password. If it detects alias duplication, then it asks you for a new alias, and you can specify a new alias or simply allow the keytool command to overwrite the existing one.

For example, to import entries from a typical JKS type keystore key.jks into a PKCS #11 type hardware-based keystore, use the command:

keytool -importkeystore

-srckeystore key.jks -destkeystore NONE

-srcstoretype JKS -deststoretype PKCS11

-srcstorepass <src keystore password>

-deststorepass <destination keystore pwd>

The importkeystore command can also be used to import a single entry from a source keystore to a destination keystore. In this case, besides the options you see in the previous example, you need to specify the alias you want to import. With the -srcalias option specified, you can also specify the destination alias name in the command line, as well as protection password for a secret/private key and the destination protection password you want. The following command demonstrates this:

keytool -importkeystore

-srckeystore key.jks -destkeystore NONE

-srcstoretype JKS -deststoretype PKCS11

-srcstorepass <src keystore password>

-deststorepass <destination keystore pwd>

-srcalias myprivatekey -destalias myoldprivatekey

-srckeypass <source entry password>

-destkeypass <destination entry password>

-noprompt

GENERATE CERTIFICATES FOR AN SSL SERVER

The following are keytool commands to generate key pairs and certificates for three entities: Root CA (root), Intermediate CA (ca), and SSL server (server). Ensure that you store all the certificates in the same keystore. In these examples, RSA is the recommended the key algorithm.

keytool -genkeypair -keystore root.jks -alias root -ext bc:c

keytool -genkeypair -keystore ca.jks -alias ca -ext bc:c

keytool -genkeypair -keystore server.jks -alias server

keytool -keystore root.jks -alias root -exportcert -rfc > root.pem

keytool -storepass <storepass> -keystore ca.jks -certreq -alias ca |

keytool -storepass <storepass> -keystore root.jks

-gencert -alias root -ext BC=0 -rfc > ca.pem

keytool -keystore ca.jks -importcert -alias ca -file ca.pem

keytool -storepass <storepass> -keystore server.jks -certreq -alias server |

keytool -storepass <storepass> -keystore ca.jks -gencert -alias ca

```
    -ext ku:c=dig,kE -rfc > server.pem
  cat root.pem ca.pem server.pem |
    keytool -keystore server.jks -importcert -alias server
```

TERMS

Keystore

A keystore is a storage facility for cryptographic keys and certificates.

Keystore entries

Keystores can have different types of entries. The two most applicable entry types for the keytool command include the following:

Key entries: Each entry holds very sensitive cryptographic key information, which is stored in a protected format to prevent unauthorized access. Typically, a key stored in this type of entry is a secret key, or a private key accompanied by the certificate chain for the corresponding public key. See Certificate Chains. The keytool command can handle both types of entries, while the jarsigner tool only handles the latter type of entry, that is private keys and their associated certificate chains.

Trusted certificate entries: Each entry contains a single public key certificate that belongs to another party. The entry is called a trusted certificate because the keystore owner trusts that the public key in the certificate belongs to the identity identified by the subject (owner) of the certificate. The issuer of the certificate vouches for this, by signing the certificate.

KeyStore aliases

All keystore entries (key and trusted certificate entries) are accessed by way of unique aliases.

An alias is specified when you add an entity to the keystore with the -genseckey command to generate a secret key, the -genkeypair command to generate a key pair (public and private key), or the -importcert command to add a certificate or certificate chain to the list of trusted certificates.

Subsequent keytool commands must use this same alias to refer to the entity.

For example, you can use the alias duke to generate a new public/private key pair and wrap the public key into a self-signed certificate with the following command. See Certificate Chains.

keytool -genkeypair -alias duke -keypass dukekeypasswd

This example specifies an initial password of dukekeypasswd required by subsequent commands to access the private key associated with the alias duke. If you later want to change Duke's private key password, use a command such as the following:

keytool -keypasswd -alias duke -keypass dukekeypasswd -new newpass

This changes the password from dukekeypasswd to newpass. A password should not be specified on a command line or in a script unless it is for testing purposes, or you are on a secure system. If you do not specify a required password option on a command line, then you are prompted for it.

KeyStore implementation

The KeyStore class provided in the java.security package supplies well-defined interfaces to access and modify the information in a keystore. It is possible for there to be multiple different concrete implementations, where each implementation is that for a particular type of keystore.

Currently, two command-line tools (keytool and jarsigner) and a GUI-based tool named Policy Tool make use of keystore implementations. Because the KeyStore class is public, users can write additional security applications that use it.

There is a built-in default implementation, provided by Oracle. It implements the keystore as a file with a proprietary keystore type (format) named JKS. It protects each private key with its individual password, and also protects the integrity of the entire keystore with a (possibly different) password.

Keystore implementations are provider-based. More specifically, the application interfaces supplied by KeyStore are implemented in terms of a Service Provider Interface (SPI). That is, there is a corresponding abstract KeystoreSpi class, also in the java.security package, which defines the Service Provider Interface methods that providers must implement. The term provider refers to a package or a set of packages that supply a concrete implementation of a subset of services that can be accessed by the Java Security API. To provide a keystore implementation, clients must implement a provider and supply a KeystoreSpi subclass implementation, as described in

How to Implement a Provider in the Java Cryptography Architecture at http://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/HowToImplAProvider.html

Applications can choose different types of keystore implementations from different providers, using the getInstance factory method supplied in the KeyStore class. A keystore type defines the storage and data format of the keystore information, and the algorithms used to protect private/secret keys in the keystore and the integrity of the keystore. Keystore implementations of different types are not compatible.

The keytool command works on any file-based keystore implementation. It treats the keystore location that is passed to it at the command line as a file name and converts it to a FileInputStream, from which it loads the keystore information.)The jarsigner command can read a keystore from any location that can be specified with a URL.

For keytool and jarsigner, you can specify a keystore type at the command line, with the -storetype option. For Policy Tool, you can specify a keystore type with the Keystore menu.

If you do not explicitly specify a keystore type, then the tools choose a keystore implementation based on the value of the keystore.type property specified in the security properties file. The security properties file is called java.security, and resides in the security properties directory, java.home\lib\security on Windows and java.home/lib/security on Oracle Solaris, where java.home is the runtime environment directory. The jre directory in the SDK or the top-level directory of the Java Runtime Environment (JRE).

Each tool gets the keystore.type value and then examines all the currently installed providers until it finds one that implements a keystores of that type. It then uses the keystore implementation from that provider.The KeyStore class defines a static method named getDefaultType that lets applications and applets retrieve the value of the keystore.type property. The following line of code creates an instance of the default keystore type as specified in the keystore.type property:

KeyStore keyStore = KeyStore.getInstance(KeyStore.getDefaultType());

The default keystore type is jks, which is the proprietary type of the

keystore implementation provided by Oracle. This is specified by the following line in the security properties file:

keystore.type=jks

To have the tools utilize a keystore implementation other than the default, you can change that line to specify a different keystore type. For example, if you have a provider package that supplies a keystore implementation for a keystore type called pkcs12, then change the line to the following:

keystore.type=pkcs12

Note: Case does not matter in keystore type designations. For example, JKS would be considered the same as jks.

Certificate

A certificate (or public-key certificate) is a digitally signed statement from one entity (the issuer), saying that the public key and some other information of another entity (the subject) has some specific value. The following terms are related to certificates:

Public Keys: These are numbers associated with a particular entity, and are intended to be known to everyone who needs to have trusted interactions with that entity. Public keys are used to verify signatures.

Digitally Signed: If some data is digitally signed, then it is stored with the identity of an entity and a signature that proves that entity knows about the data. The data is rendered unforgeable by signing with the entity's private key.

Identity: A known way of addressing an entity. In some systems, the identity is the public key, and in others it can be anything from an Oracle Solaris UID to an email address to an X.509 distinguished name.

Signature: A signature is computed over some data using the private key of an entity. The signer, which in the case of a certificate is also known as the issuer.

Private Keys: These are numbers, each of which is supposed to be known only to the particular entity whose private key it is (that is, it is supposed to be kept secret). Private and public keys exist in pairs in all public key cryptography systems (also referred to as public key crypto systems). In a typical public key crypto system, such as DSA, a private key corresponds to

exactly one public key. Private keys are used to compute signatures.

Entity: An entity is a person, organization, program, computer, business, bank, or something else you are trusting to some degree.

Public key cryptography requires access to users' public keys. In a large-scale networked environment, it is impossible to guarantee that prior relationships between communicating entities were established or that a trusted repository exists with all used public keys. Certificates were invented as a solution to this public key distribution problem. Now a Certification Authority (CA) can act as a trusted third party. CAs are entities such as businesses that are trusted to sign (issue) certificates for other entities. It is assumed that CAs only create valid and reliable certificates because they are bound by legal agreements. There are many public Certification Authorities, such as VeriSign, Thawte, Entrust, and so on.

You can also run your own Certification Authority using products such as Microsoft Certificate Server or the Entrust CA product for your organization. With the keytool command, it is possible to display, import, and export certificates. It is also possible to generate self-signed certificates.

The keytool command currently handles X.509 certificates.

X.509 Certificates

The X.509 standard defines what information can go into a certificate and describes how to write it down (the data format). All the data in a certificate is encoded with two related standards called ASN.1/DER. Abstract Syntax Notation 1 describes data. The Definite Encoding Rules describe a single way to store and transfer that data.

All X.509 certificates have the following data, in addition to the signature:

Version: This identifies which version of the X.509 standard applies to this certificate, which affects what information can be specified in it. Thus far, three versions are defined. The keytool command can import and export v1, v2, and v3 certificates. It generates v3 certificates.

X.509 Version 1 has been available since 1988, is widely deployed, and is

the most generic.

X.509 Version 2 introduced the concept of subject and issuer unique identifiers to handle the possibility of reuse of subject or issuer names over time. Most certificate profile documents strongly recommend that names not be reused and that certificates should not make use of unique identifiers. Version 2 certificates are not widely used.

X.509 Version 3 is the most recent (1996) and supports the notion of extensions where anyone can define an extension and include it in the certificate. Some common extensions are: KeyUsage (limits the use of the keys to particular purposes such as signing-only) and AlternativeNames (allows other identities to also be associated with this public key, for example. DNS names, email addresses, IP addresses). Extensions can be marked critical to indicate that the extension should be checked and enforced or used. For example, if a certificate has the KeyUsage extension marked critical and set to keyCertSign, then when this certificate is presented during SSL communication, it should be rejected because the certificate extension indicates that the associated private key should only be used for signing certificates and not for SSL use.

Serial number: The entity that created the certificate is responsible for assigning it a serial number to distinguish it from other certificates it issues. This information is used in numerous ways. For example, when a certificate is revoked its serial number is placed in a Certificate Revocation List (CRL).

Signature algorithm identifier: This identifies the algorithm used by the CA to sign the certificate.

Issuer name: The X.500 Distinguished Name of the entity that signed the certificate. See X.500 Distinguished Names. This is typically a CA. Using this certificate implies trusting the entity that signed this certificate.

In some cases, such as root or top-level CA certificates, the issuer signs its own certificate.

Validity period: Each certificate is valid only for a limited amount of time. This period is described by a start date and time and an end date and time, and can be as short as a few seconds or almost as long as a century.

The validity period chosen depends on a number of factors, such as the strength of the private key used to sign the certificate, or the amount one is willing to pay for a certificate. This is the expected period that entities can rely on the public value, when the associated private key has not been compromised.

Subject name: The name of the entity whose public key the certificate identifies. This name uses the X.500 standard, so it is intended to be unique across the Internet. This is the X.500 Distinguished Name (DN) of the entity. See X.500 Distinguished Names. For example,

CN=Java Duke, OU=Java Software Division, O=Oracle Corporation, C=US

These refer to the subject's common name (CN), organizational unit (OU), organization (O), and country (C).

Subject public key information: This is the public key of the entity being named with an algorithm identifier that specifies which public key crypto system this key belongs to and any associated key parameters.

Certificate Chains

The keytool command can create and manage keystore key entries that each contain a private key and an associated certificate chain. The first certificate in the chain contains the public key that corresponds to the private key.

When keys are first generated, the chain starts off containing a single element, a self-signed certificate. See -genkeypair in Commands. A self-signed certificate is one for which the issuer (signer) is the same as the subject. The subject is the entity whose public key is being authenticated by the certificate. Whenever the -genkeypair command is called to generate a new public/private key pair, it also wraps the public key into a self-signed certificate.

Later, after a Certificate Signing Request (CSR) was generated with the -certreq command and sent to a Certification Authority (CA), the response from the CA is imported with -importcert, and the self-signed certificate is replaced by a chain of certificates. See the -certreq and -importcert options in Commands. At the bottom of the chain is the certificate (reply) issued by the CA authenticating the subject's public key. The next

certificate in the chain is one that authenticates the CA's public key.
In many cases, this is a self-signed certificate, which is a certificate from the CA authenticating its own public key, and the last certificate in the chain. In other cases, the CA might return a chain of certificates. In this case, the bottom certificate in the chain is the same (a certificate signed by the CA, authenticating the public key of the key entry), but the second certificate in the chain is a certificate signed by a different CA that authenticates the public key of the CA you sent the CSR to. The next certificate in the chain is a certificate that authenticates the second CA's key, and so on, until a self-signed root certificate is reached. Each certificate in the chain (after the first) authenticates the public key of the signer of the previous certificate in the chain.
Many CAs only return the issued certificate, with no supporting chain, especially when there is a flat hierarchy (no intermediates CAs). In this case, the certificate chain must be established from trusted certificate information already stored in the keystore.
A different reply format (defined by the PKCS #7 standard) includes the supporting certificate chain in addition to the issued certificate. Both reply formats can be handled by the keytool command.
The top-level (root) CA certificate is self-signed. However, the trust into the root's public key does not come from the root certificate itself, but from other sources such as a newspaper. This is because anybody could generate a self-signed certificate with the distinguished name of, for example, the VeriSign root CA. The root CA public key is widely known. The only reason it is stored in a certificate is because this is the format understood by most tools, so the certificate in this case is only used as a vehicle to transport the root CA's public key. Before you add the root CA certificate to your keystore, you should view it with the -printcert option and compare the displayed fingerprint with the well-known fingerprint obtained from a newspaper, the root CA's Web page, and so on.

The cacerts Certificates File

A certificates file named cacerts resides in the security properties directory, java.home\lib\security on Windows and java.home/lib/security on

Oracle Solaris, where java.home is the runtime environment's directory, which would be the jre directory in the SDK or the top-level directory of the JRE.

The cacerts file represents a system-wide keystore with CA certificates. System administrators can configure and manage that file with the keytool command by specifying jks as the keystore type. The cacerts keystore file ships with a default set of root CA certificates. You can list the default certificates with the following command:

keytool -list -keystore java.home/lib/security/cacerts

The initial password of the cacerts keystore file is changeit. System administrators should change that password and the default access permission of that file upon installing the SDK.

Note: It is important to verify your cacerts file. Because you trust the CAs in the cacerts file as entities for signing and issuing certificates to other entities, you must manage the cacerts file carefully. The cacerts file should contain only certificates of the CAs you trust. It is your responsibility to verify the trusted root CA certificates bundled in the cacerts file and make your own trust decisions.

To remove an untrusted CA certificate from the cacerts file, use the delete option of the keytool command. You can find the cacerts file in the JRE installation directory. Contact your system administrator if you do not have permission to edit this file

Internet RFC 1421 Certificate Encoding Standard

Certificates are often stored using the printable encoding format defined by the Internet RFC 1421 standard, instead of their binary encoding. This certificate format, also known as Base64 encoding, makes it easy to export certificates to other applications by email or through some other mechanism.

Certificates read by the -importcert and -printcert commands can be in either this format or binary encoded. The -exportcert command by default outputs a certificate in binary encoding, but will instead output a certificate in the printable encoding format, when the -rfc option is specified.

The -list command by default prints the SHA1 fingerprint of a certificate.

If the -v option is specified, then the certificate is printed in human-readable format. If the -rfc option is specified, then the certificate is output in the printable encoding format.

In its printable encoding format, the encoded certificate is bounded at the beginning and end by the following text:

-----BEGIN CERTIFICATE-----

encoded certificate goes here.

-----END CERTIFICATE-----

X.500 Distinguished Names

X.500 Distinguished Names are used to identify entities, such as those that are named by the subject and issuer (signer) fields of X.509 certificates.

The keytool command supports the following subparts:

commonName: The common name of a person such as Susan Jones.

organizationUnit: The small organization (such as department or division) name. For example, Purchasing.

localityName: The locality (city) name, for example, Palo Alto.

stateName: State or province name, for example, California.

country: Two-letter country code, for example, CH.

When you supply a distinguished name string as the value of a -dname option, such as for the -genkeypair command, the string must be in the following format:

CN=cName, OU=orgUnit, O=org, L=city, S=state, C=countryCode

All the italicized items represent actual values and the previous keywords are abbreviations for the following:

CN=commonName

OU=organizationUnit

O=organizationName

L=localityName

S=stateName

C=country

A sample distinguished name string is:

CN=Mark Smith, OU=Java, O=Oracle, L=Cupertino, S=California, C=US

A sample command using such a string is:

keytool -genkeypair -dname "CN=Mark Smith, OU=Java, O=Oracle, L=Cupertino, S=California, C=US" -alias mark

Case does not matter for the keyword abbreviations. For example, CN, cn, and Cn are all treated the same.

Order matters; each subcomponent must appear in the designated order. However, it is not necessary to have all the subcomponents. You can use a subset, for example:

CN=Steve Meier, OU=Java, O=Oracle, C=US

If a distinguished name string value contains a comma, then the comma must be escaped by a backslash (\) character when you specify the string on a command line, as in:

cn=Peter Schuster, ou=Java\, Product Development, o=Oracle, c=US

It is never necessary to specify a distinguished name string on a command line. When the distinguished name is needed for a command, but not supplied on the command line, the user is prompted for each of the subcomponents. In this case, a comma does not need to be escaped by a backslash (\).

WARNINGS

IMPORTING TRUSTED CERTIFICATES WARNING

Important: Be sure to check a certificate very carefully before importing it as a trusted certificate.

Windows Example:

View the certificate first with the -printcert command or the -importcert command without the -noprompt option. Ensure that the displayed certificate fingerprints match the expected ones. For example, suppose sends or emails you a certificate that you put it in a file named \tmp\cert. Before you consider adding the certificate to your list of trusted certificates, you can execute a -printcert command to view its fingerprints, as follows:

keytool -printcert -file \tmp\cert

Owner: CN=ll, OU=ll, O=ll, L=ll, S=ll, C=ll

Issuer: CN=ll, OU=ll, O=ll, L=ll, S=ll, C=ll

Serial Number: 59092b34

Valid from: Thu Sep 25 18:01:13 PDT 1997 until: Wed Dec 24 17:01:13 PST 1997

Certificate Fingerprints:

MD5: 11:81:AD:92:C8:E5:0E:A2:01:2E:D4:7A:D7:5F:07:6F

SHA1: 20:B6:17:FA:EF:E5:55:8A:D0:71:1F:E8:D6:9D:C0:37:13:0E:5E:FE

SHA256: 90:7B:70:0A:EA:DC:16:79:92:99:41:FF:8A:FE:EB:90:

17:75:E0:90:B2:24:4D:3A:2A:16:A6:E4:11:0F:67:A4

Oracle Solaris Example:

View the certificate first with the -printcert command or the -importcert command without the -noprompt option. Ensure that the displayed certificate fingerprints match the expected ones. For example, suppose someone sends or emails you a certificate that you put it in a file named /tmp/cert. Before you consider adding the certificate to your list of trusted certificates, you can execute a -printcert command to view its fingerprints, as follows:

keytool -printcert -file /tmp/cert

Owner: CN=ll, OU=ll, O=ll, L=ll, S=ll, C=ll

Issuer: CN=ll, OU=ll, O=ll, L=ll, S=ll, C=ll

Serial Number: 59092b34

Valid from: Thu Sep 25 18:01:13 PDT 1997 until: Wed Dec 24 17:01:13 PST 1997

Certificate Fingerprints:

MD5: 11:81:AD:92:C8:E5:0E:A2:01:2E:D4:7A:D7:5F:07:6F

SHA1: 20:B6:17:FA:EF:E5:55:8A:D0:71:1F:E8:D6:9D:C0:37:13:0E:5E:FE

SHA256: 90:7B:70:0A:EA:DC:16:79:92:99:41:FF:8A:FE:EB:90:

17:75:E0:90:B2:24:4D:3A:2A:16:A6:E4:11:0F:67:A4

Then call or otherwise contact the person who sent the certificate and compare the fingerprints that you see with the ones that they show. Only when the fingerprints are equal is it guaranteed that the certificate was not replaced in transit with somebody else's certificate such as an attacker's certificate. If such an attack took place, and you did not check the certificate before you imported it, then you would be trusting anything the attacker signed, for example, a JAR file with malicious class files inside.

Note: It is not required that you execute a -printcert command before importing a certificate. This is because before you add a certificate to the list of trusted certificates in the keystore, the -importcert command prints out the certificate information and prompts you to verify it. You can then stop the import operation. However, you can do this only when you call the -importcert command without the

-noprompt option. If the -noprompt option is specified, then there is no interaction with the user.

PASSWORDS WARNING

Most commands that operate on a keystore require the store password. Some commands require a private/secret key password. Passwords can be specified on the command line in the -storepass and -keypass options. However, a password should not be specified on a command line or in a script unless it is for testing, or you are on a secure system. When you do not specify a required password option on a command line, you are prompted for it.

CERTIFICATE CONFORMANCE WARNING

The Internet standard RFC 5280 has defined a profile on conforming X.509 certificates, which includes what values and value combinations are valid for certificate fields and extensions. See the standard at http://tools.ietf.org/rfc/rfc5280.txt

The keytool command does not enforce all of these rules so it can generate certificates that do not conform to the standard. Certificates that do not conform to the standard might be rejected by JRE or other applications. Users should ensure that they provide the correct options for -dname, -ext, and so on.

NOTES

IMPORT A NEW TRUSTED CERTIFICATE

Before you add the certificate to the keystore, the keytool command verifies it by attempting to construct a chain of trust from that certificate to a self-signed certificate (belonging to a root CA), using trusted certificates that are already available in the keystore.

If the -trustcacerts option was specified, then additional certificates are considered for the chain of trust, namely the certificates in a file named cacerts.

If the keytool command fails to establish a trust path from the certificate to be imported up to a self-signed certificate (either from the keystore or the cacerts file), then the certificate information is printed, and the user is prompted to verify it by comparing the displayed certificate fingerprints with the fingerprints obtained from some other (trusted) source of information, which might be the certificate owner. Be very careful to ensure the certificate is valid before importing it as a trusted certificate. See Importing Trusted Certificates Warning.

The user then has the option of stopping the import operation. If the -noprompt option is specified, then there is no interaction with the user.

IMPORT A CERTIFICATE REPLY

When you import a certificate reply, the certificate reply is validated with trusted certificates from the keystore, and optionally, the certificates configured in the cacerts keystore file when the -trustcacerts option is specified. See The cacerts Certificates File.

The methods of determining whether the certificate reply is trusted are as follows:

? If the reply is a single X.509 certificate, then the keytool command attempts to establish a trust chain, starting at the certificate reply and ending at a self-signed certificate (belonging to a root CA). The certificate reply and the hierarchy of certificates is used to authenticate the certificate reply from the new certificate chain of aliases. If a trust chain cannot be established, then the certificate reply is not imported. In this case, the keytool command does not print the certificate and prompt the user to verify it, because it is very difficult for a user to determine the authenticity of the certificate reply.

? If the reply is a PKCS #7 formatted certificate chain or a sequence of X.509 certificates, then the chain is ordered with the user certificate first followed by zero or more CA certificates. If the chain ends with a self-signed root CA certificate and the-trustcacerts option was specified, the keytool command attempts to match it with any of the trusted certificates in the keystore or the cacerts keystore file. If the chain does not end with a self-signed root CA certificate and the -trustcacerts option was specified, the keytool command tries to find one from the trusted certificates in the keystore or the cacerts keystore file and add it to the end of the chain. If the certificate is not found and the -noprompt option is not specified, the information of the last certificate in the chain is printed, and the user is prompted to verify it.

If the public key in the certificate reply matches the user's public key already stored with alias, then the old certificate chain is replaced with the new certificate chain in the reply. The old chain can only be replaced with a valid keypass, and so the password used to protect the private key of the entry is supplied. If no password is provided, and the private key password is different from the keystore password, the user is prompted for it.

This command was named -import in earlier releases. This old name is still supported in this release. The new name, -importcert, is preferred going forward.

SEE ALSO

? jar(1)

? jarsigner(1)

? Trail: Security Features in Java SE at http://docs.oracle.com/javase/tutorial/security/index.html

JDK 8 03 March 2015 keytool(1)